N89 - 15602

Ada in AI or AI in Ada? On Developing A Rationale For Integration

Philippe E. Collard
California Space Institute
A-016, UCSD, La Jolla, Ca 92093

Andre Goforth
NASA/Ames Research Center
Information Sciences Division, MS244-4
Moffett Field, Ca 94035

Abstract

The use of Ada as an Artificial Intelligence (AI) language has been gaining interest within the NASA community. This interest is held by parties in NASA who have a need to deploy Knowledge Based-Systems (KBS) compatible with the use of Ada as the software standard for the Space Station.

A fair number of KBS and pseudo-KBS implementations in Ada exist today. Currently, no widely used guidelines exist to compare and evaluate these with one another. The lack of such guidelines illustrates a fundamental problem inherent in trying to compare and evaluate implementations of any sort in languages that are procedural or imperative in style, such as Ada, with those in languages that are functional in style, such as Lisp.

This paper discusses the strengths and weakness of using Ada as an AI language and provides a preliminary analysis of factors needed for the development of criteria for the integration of these two families of languages and the environments in which they are implemented.

The intent for developing such criteria is to have a logical rationale that may be used to guide the development of Ada tools and methodology to support KBS requirements, and to identify those AI technology components that may most readily and effectively be deployed in Ada versus those best left in a functional language.

INTRODUCTION

The AI community and the Ada software engineering community have historically evolved and developed methodologies and formalisms with only a modest amount of collaboration to date. This is partly due to their respective goals. On one hand, the AI community has been concerned with the "what" and "why" of a problem, *i.e.*, the representation of information indigenous to the development of requirements. On the other hand, the Ada software engineering community has concerned itself more with the "how to" or methodology of correctly arriving at a system implementation that may have little in common with the knowledge used in generating the original requirements.

By mandate, NASA has made Ada the *de facto* language for the Space Station. The authors believe this decision was rightfully made on the grounds that Ada and the software engineering principles it embodies will be an invaluable assets in keeping development and maintenance costs manageable at all levels. In the past ten years, Ada has made significant progress in realizing the expectations of its designers. There is merit, however, in the intuitive, and often demonstrated, notion that there is no "silver bullet" [1] in computer sciences. There are and probably will continue to be, difficulties in using a single tool for the implementation of all the components of Space Station software, be they AI systems or not.

A major issue confronting those parties who have a need to deploy both ground and in-flight KBS systems is: *how is this mandate to be interpreted or executed?* For example, policy makers may view the mandate as allowing for three alternatives in hosting AI technology in the Space Station software environment:

• Ada as the only host language. In this alternative, regardless of any other considerations, Ada is the only allowable host environment. Any AI technology to be used in the Space Station must, therefore, be made to operate in Ada.

• Ada as a co-host language. Common Lisp (and other desirable AI languages) is accommodated in the Space Station environment with Ada through the use of standard interfaces and the implementation of common development tools and maintenance procedures.

• Ada as a temporary co-host language. Common Lisp is accommodated as described above, but with the condition that, over a certain period of time, all non-Ada code must gradually be phased into an Ada implementation. There may be other variations of this case that may be of interest to policy makers and user advocates.

The mandate interpretation issue applies to implementors and users of other specialized computer technology areas that are perceived as being outside the current capability envelope of Ada implementation. For example, real-time requirements of some subsystems and the database search and retrieval requirements of some applications may have non-Ada "off the shelf" solutions which come with lower risk factors for meeting a required capability, resource, and schedule envelope. However, in this case, the risk factor re-surfaces in the effort required to integrate , verify and maintain the "non-native" solution with the host Ada environment.

The dilemma of local or project-specific optimization, in terms of use of a special technology, versus the strategic or global system optimization, in terms of a mandated technology choice, is inevitable. The matter of making exceptions, *i.e.* waivers, to a policy might be viewed as detrimental to the intent and execution of the policy. However, it may be argued that some compromise is necessary in these circumstances. The general nature of the problem is brought out here because a solution to it will have an impact on the solution to the specific problem at hand; *i.e.*. how to accommodate AI technology and "culture" in the Space Station software environment.

A recommendation for resolving this dilemma is beyond the scope of this paper. The scope is limited to discussing the technical issues of accommodating AI technology and "culture" in Ada from a managerial view. The purpose of this discussion is to enhance the level of understanding of the issues and factors that need to be explored more thoroughly for developing effective integration criteria for Ada as an AI language in the Space Station.

The advantages of Ada for the implementation of Space Station AI systems are reviewed in the following discussion.

AI IN ADA: STRENGTHS

a) Ada is a standard

Ada is now an ANSI and ISO standard. Its definition is clearly stated by the Language Reference Manual (LRM). Unfortunately, the validation suite does not address performance issues, but it does ensure that an Ada compiler delivers objects that conform to the syntax and semantics defined by the LRM. If some latitude is given to the implementors by Chapter 13 of the LRM, the variations from compiler to compiler are limited to specific features, and must be clearly documented. All in all, the availability of such a standard definition allows for a smoother development cycle by increasing the portability and re-usability of software components, facilitating maintenance, and reducing the training cost. All of these things will benefit the development of the Space Station AI applications and their interfacing with other components of the Space Station software.

Standard definitions for AI languages, even the most commonly used such as Lisp and Prolog, are not as well developed as for Ada, though efforts are underway to unify the various dialects of Lisp into Common Lisp. For Prolog, the definition given by Clocksin and Mellish [2] is the closest to what may be called a widely accepted standard.

This lack of standardization can only be detrimental to the development of large, "real life" projects such as the Space Station. This conclusion, reached by the DoD community after studying the consequences of the proliferation of programming languages and dialects providing a rationale for the development of Ada [3]. Having a standard that is certifiable by an independent means is very important for gaining Agency acceptance.

b) Ada favors the use of modern software engineering techniques

Ada is not only a programming language but also a methodology of development for large software projects. Ada was designed for "programming in the large". By contrast, software engineering has never been a predominant concern in the AI community. One reason for that is the fact that the implementation of AI systems has traditionally been based on rapid prototyping and incremental development - initially an appropriate approach because it was well suited to the size of the applications being developed. As stated in the introduction, software engineers focus on the "how to", whereas knowledge engineers focus on "what" and "why". The two groups use different tools and methodology.

Some may argue that the concept of "engineering" is contrary to the dynamic nature of AI systems. After all, these systems exhibit features, such as self-modifying code, that are considered harmful in light of the current software engineering principles. On the contrary, it has been demonstrated that AI work and software engineering are, indeed, complementary and not orthogonal [4].

To reach today's expectations, Space Station AI systems will have to be of a size and scope that have no current equal. Furthermore, they will have to be highly reliable and maintainable. It may be questioned whether existing iterative processes used to develop today's AI systems can be directly scaled to accommodate the increased size and scope. The rationale for questioning and caution is based on the past experience in the AI field that AI techniques, in some cases, have not scaled-up effectively. Incorporating the experience and methodology of the Ada community may greatly aid in the scaling-up of AI systems in the Space Station.

c) Ada is available on space qualified hardware/software platforms

Because of its special sponsorship by the U.S. Government, Ada is becoming available on an increasing number of hardware and software platforms suitable for space flight. Because of its standard definition, applications developed on one of these space-qualified platforms can easily be ported to another platform, provided they avoid making references to system-dependant features (as defined by Chapter 13 of the LRM). This allows for greater flexibility in the development process.

d) Ada's tasking construct is well suited to AI applications

Although Ada was not initially designed for AI, it contains some constructs that are well suited or readily adapted to AI work. The most notable example is tasking, giving Ada the ability to support parallel processing at the language level. Whether it is for fine-grained parallelism [5] or for large-grained parallelism [6], the availability of the tasking construct allows for the development of AI systems in terms of concurrent execution of computing or inferring units without requiring any radical extension or modification of the language.

e) The use of Ada can improve the development of real-time expert systems

In the context of the Space Station, on-board AI applications will have to respond in real-time or near real-time. The status of real-time expert systems is far from being adequate. A survey of this field recently published in AI Magazine[7] , a publication of the AAAI, concluded that: "current expert system shells are two or three orders of magnitude too slow", "current shells cannot guarantee response times", "current shells have little or no capabilities for temporal reasoning", "current shells lack the facilities to handle hardware and software interrupts". All these features, currently lacking in available AI tools, are critical to the implementation of Space Station systems.

Real-time expert systems is another area where the use of Ada may be extremely positive. Ada was designed specifically for embedded real-time systems. Although the suitability of Ada real-time constructs is open to debate, it is undeniable that the use of Ada for real-time systems is responsible for major advances in this field. First, real-time systems can be designed and coded more cleanly using software engineering techniques. Second, the language supports real-time features such as tasking and exceptions and hardware interrupt handling without extension or references to any particular operating system. Lastly, significant progress is being made in the area of run-time systems to support Ada on embedded targets. Thus, one of the consequences of using Ada for building real-time expert systems is benefit from these advances and, therefore, a remedy some of the problems encountered with current AI tools.

f) Ada may be used as a standard PDL for AI applications

Even if Ada was not used for the implementation of some Space Station AI systems or subsystems, it could be used as Program Design Language for these same systems. This will ensure a commonality in the communication media between the various development teams and therefore would facilitate interfacing and integration.

The following discussion reviews the limitations of Ada with regard to the implementation of Space Station AI systems.

AI IN ADA: WEAKNESSES

a) Rapid prototyping is difficult in Ada

Rapid prototyping is a trademark of AI development and requires tools with great flexibility. There are two major drawbacks to the use of Ada for building prototypes of AI applications. First, Ada is a compiled language. The LRM defines rather strict rules regarding what is to be re-compiled when a unit of the compilation library is modified. These rules may imply that large portions of a system may have to be re-compiled after just one modification. This puts a heavy burden on the prototyping work. Unlike interpreters, compilers are not well adapted to interactive development. They are designed to produce efficient object modules, whereas interpreters are designed to allow quick and easy modification and testing of programs. However, the development of incremental compilers may help reduce the overhead imposed by compilations and re-compilations.

The strong typing rules of Ada constitute the second obstacle to using Ada for AI prototyping. With Ada, every variable must be declared of a certain type. The iterative and dynamic nature of the prototyping work would be likely to cause frequent changes in the typing scheme, requiring type declarations to be traced and modified throughout the program. This could add significantly to the overhead as well as multiply the causes for error.

b) Technical difficulties

There are a number of linguistic technical problems that limit the use of Ada for AI applications. Depending on the specificity of each application, the impact of each is felt differently. For example, it is much easier to use Ada for developing an expert system shell than for doing symbolic computation. Some of the major technical problems include:

1) Storage management and lack of garbage collection. AI applications are prone to high consumption of dynamic storage. The definition of Ada does not require an implementor to provide garbage collection facilities. In addition, Ada does not allow for static pointers (pointers to objects that are already declared).

2) Early binding. Binding decisions are made early in Ada. This is not the case for most AI languages. Early binding reduces, and sometimes prevents, the possibility of dynamically defining new types as required by the evolving context of the execution (for instance, by creating new data types at run-time).

3) Functions and procedures cannot be arguments of other functions or procedures. This prevents self-modifying code from being implemented in Ada. Therefore, it is more difficult for an AI application written in Ada to adapt to a changing context than for the same application written in Lisp. More generally, Ada lacks the dynamic constructs that are commonly found in most AI languages.

c) Cultural and training issues

G. Booch, a noted Ada expert, noted that "a programming language shapes the way we think about a solution. We need a language that leads to systems that map directly to their problem space" [8]. This is where "cultural" differences appear between the AI community and the software engineering community. One concerns understanding the theory of computation, the other concerns building efficient and reliable computational engines. This has lead to a divergence in design and evolution of the tools used by each group. Using Ada for developing AI systems implies that each community will have to understand to a certain extent and, adopt the cultural background of the other to merge it with its own.

A NEED FOR ADDITIONAL FACTORS

Developing a list of Ada's strengths and weaknesses as a development and implementation environment for AI is a step toward formulating a rationale for integration. A more extensive and exhaustive study may uncover additional strengths and weaknesses. Weights could then be assigned to each of these, according to some priority scheme, and the choice made, based on the relative total weights. This, however, provides an incomplete picture. First, given such a short list where every point is pivotal, arriving at a rational weighting scheme may prove to be intractable or unacceptably arbitrary. Second, the possibility of Ada as a co-host is, for the most part, overlooked since this could only be covered by adopting an arbitrary scoring range to represent it as a choice. Therefore, additional factors may have to be considered in developing an integration rationale.

An additional factor to consider is the development of evaluation criteria composed of significant software engineering factors. A list of such factors, not to be construed as exhaustive, is given below:

• Project Size For large scale projects, one must evaluate which solution provides the best way to handle all the problems related to size.

• Training Is there a readily available pool of trained engineers who can make the solution work for all aspects of the projects?

• Performance What is the solution that is most likely to provide better performance in the short- , mid-, and long-term?

• Interfacing to other environments Which solution will be the easiest to interface to other systems, subsystems, or environments ?

• Evolution Which solution will provide a smoother evolutionary path?

• Maintenance Which solution will facilitate maintenance best?

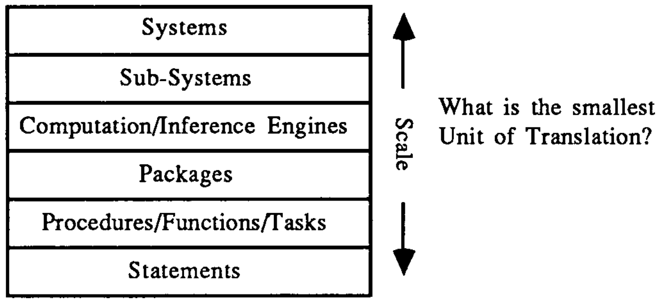
• Verification and Validation Is there a solution that makes Verification and Validation easier to perform ?

• Productivity tools Which solution provides the better software productivity tools and life cycle environment?

• Real-time accommodation For applications that require real-time or near real-time performance, which solution is likely to perform better?

Other factors to consider are the various approaches to using Ada as an AI language. Emulation of Lisp features in Ada with special packages and techniques is one possibility. To better understand these and other approaches, it is useful to attempt to categorize them in some uniform and constructive manner. Any attempt to replicate the behavior of an application written in Lisp in Ada depends on the scale at which a translation is performed. The scale is illustrated in the figure below as a hierarchy of levels in the Space Station software environment.

Space Station Software Environment Levels


Systems
Sub-Systems
Computation/Inference Engines
Packages
Procedures/Functions/Tasks
Statements
Scale
What is the smallest Unit of Translation?

The figure above shows the different levels at which the insertion of AI technology may be made in the Space Station software environment. Any approach to implementing AI technology must enter at some level of the hierarchy, though it may cover more than one level. Since there is no accepted or standardized way to categorize all of the approaches for using Ada as an AI language, the following categorization is put forward based on information currently available in the literature:

• Automatic Translation to Host Environment
• Emulation of features in Host Environment
• Interface/Front-end Processing to Host Environment
• Host Implementation From Requirements Specification

The boundaries between these are not sharp, and some AI in Ada approaches may be viewed as hybrids of these. The issues of the suitability of using Ada for AI was raised early in the life of Ada[9]; however, the topic has not been explored as nearly as much as other Ada issues, such as, real-time support. Still, the literature available on the topic allows for a preliminary assessment of the approaches. In all likelihood, the research and development of this Ada topic will burgeon enormously in the very near future. A number of the publications available to date are listed in the references for this paper [10],[11],[12],[13],[14]. In addition, under the auspices of George Mason University, Fairfax Va., in cooperation with the Software Productivity Consortium, the AIDA conference is held annually and serves as a forum for parties interested in studying Ada and Artificial Intelligence.

The most straightforward way to use Ada as an AI language is to use an automatic translator. Lisp code is input to a translator that produces Ada code which is then compiled and executed. Ideally, the results of the Ada version are identical to those found in the Lisp version.

At present, the automatic translation approach is being applied primarily to the Statements layer. Due to the large semantic gaps between Lisp, a functional language, and Ada, a procedural language, this approach has, so far, been limited to restricted subsets of Lisp. The linguistic issues have been discussed in the literature[12],[15].

Emulation of Lisp features in Ada is closely allied with translation at the Statements Level. Although this approach enters at a higher level of the hierarchy than the automatic translation approach, the boundary between the two is not sharp. This approach may cover the next two higher levels in the hierarchy, the Procedures/Functions/Tasks and Packages. With this approach, libraries of packages are provided to the Ada programmer with which to fashion the emulation of many list processing types of constructions. One such approach was reported by Reeker and Wauchope in 1986 [16].

Interface, or front-end processing is directed mainly at the Sub-systems and Computation/Inference Engines level. Here too, the boundaries are not sharp. In this type of approach the AI user is presented with a familiar expert systems shell format that is implemented all or partly in a non-Ada language and environment. The results are then processed into the Ada environment. In this approach, the critical link is the specification of the interface that performs the conversion/translation of knowledge-based representation into Ada data types.

Host implementation from requirements specification is the highest level of all the approaches in that it is directed at the Systems and Sub-systems level. A requirements specification of an AI application, such as an expert system shell, is generated and then used to implement, from scratch, a working version in Ada.

In any analysis of the relative strengths and weaknesses of these approaches, one overriding element that may be difficult for either side to fully appreciate is "culture." In the context of Ada and AI, culture may be defined as the priorities, the *raison d'etre* or collective principals, that serve to guide the constituents in how they accomplish their goal(s). Many capabilities that cannot be merely identified as a piece of code are potentially lost in performing any one of these approaches of using Ada as an AI language. As mentioned earlier, rapid prototyping is a hallmark of the AI community. Yet, it is difficult to convey to someone who is familiar only with Ada and its software engineering principals just what it all means, other than it is possible to accomplish a prototype and, possibly, a full scale development in a certain (record breaking) time.

Due to the necessary limitations on the length of this paper, discussion of the application of the system engineering factors, mentioned previously, to each of these approaches is limited to discussing only those criteria salient for the case of automatic translation. This discussion illustrates another step in developing an integration criteria.

Currently, commercial products exist wherein Common Lisp is compiled into C and then this code is compiled into machine code to be run on the desired host. However, there are software engineering issues that are not fully addressed with this approach even if a translator could be built that would be capable of *merely* translating correctly all of the features and language constructs occurring in Common Lisp into Ada code. They are as follows:

• Performance. Whether the application code translated into Ada should perform as fast or as slow as its implementation in Lisp ?

• Real-time Accommodation. For translation of real-time codes it appears highly unlikely that such a translator could insure the same real-time characteristics found in the Lisp version in the Ada version.

• Evolution. If new features are incorporated in the Common Lisp standard, then the translator must be updated and reverified. This task may be equal in cost to the one of building the original translator.

• Maintenance. The Ada code compiled by the translator is most likely indecipherable by software engineers unless it is built to provide comments on the translation process. Indeed, this may be equal in difficulty to building a translator that merely translates. Thereby the cost is doubled. As a consequence, all Ada code generated by the translator without built-in commentary will be dependent on the translation step. That is, any changes due to updates or due to fixing errors in the Ada version will require going back to the Lisp version of the code and making the change there, then verifying the correctness there, and then performing the translation. Therefore, the maintenance cycle for codes translated mechanically will always be hostage to the translator and the Lisp implementation of the code.

These engineering issues are less of a problem if Ada is the "native" or systems programming language of the host machine. An example of this occurs with the programming language C in which 95 percent or more of the operating system UNIX is written. The case of Ada being the systems programming language for a operating system on the scale of UNIX is a question of availability. Implementing a general purpose operating system in Ada and translating Common Lisp into efficient Ada code are technology areas that appear to need more development.

SUMMARY

Some of the factors needed in developing an integration criteria have been discussed in this paper. No conclusions are to be drawn from the discussion because the relative importance of many of the factors is open to yet another degree of scrutiny. Even if a concerted effort was mounted to once and for all answer what is the best choice - there are possibly several equally good alternatives - it is highly unlikely that it would adequately reflect the needs of all the interested parties and the evolutionary processes rapidly at work in both the Artificial Intelligence community and the Ada community. The one common denominator necessary in whatever choice is made is *training.*

There are currently relatively few people trained in both Ada software engineering and AI. This is a significant problem for the design and implementation of the Knowledge-Based Systems that are envisioned for the Space Station. For "AI in Ada" or "Ada in AI" to flourish, Ada software engineers will have to understand knowledge engineering techniques and their rapidly evolving definitions. AI specialists will have to take into account the requirements imposed by large, long-lived projects where definitions and techniques must be stabilized early in the development process. With this cross-cultural understanding, the question "Ada in AI or AI in Ada? " will become a moot point.

REFERENCES

[1] F. Brooks, No Silver Bullet: Essence and Accidents of Software Engineering, IEEE Computer, April 1987

[2] W.F. Clocksin and C.S. Mellish, Programming in Prolog, Springer-Verlag, 1981

[3] Requirements for High Order Programming Languages, IRONMAN, Department of Defense, January 1977

[4] G. Karam, An Icon-Based Design Method for Prolog, IEEE Software, July 1988

[5] A. Brintzehoff, S. Christensen, J. Mangan, J. Greco, The Use of Ada Concurrent Processing Features in an Implementation of Parallel Tree Searching Algorithms, Proceedings of AIDA-87, Third Annual Conference on Artificial Intelligence and Ada, Washington D.C., 1987

[6] R. Volz, P. Krishnan, R. Theriault, An Approach to Distributed Execution of Ada Programs, NASA Workshop on Telerobotics, January 1987

[7] T. Laffey, P. Cox & al., Real-Time Knowledge Based Systems, AI Magazine, Spring 1988

[8] G. Booch, Software Engineering with Ada, 2nd ed., Benjamin Cummings, 1987

[9] R. Schwartz, P. Melliar-Smith, On the suitability of Ada for Artificial Intelligence Applications, Project 1019, SRI, 1980

[10] P.J. Wallis, Automatic Language Conversion and its Place in the Transition to Ada, Proceedings of the Ada International Conference, Paris, 1985

[11] A. Rude, Translating A Research Lisp Prototype to A Formal Ada Design Prototype, Proceedings of the Washington Ada Symposium, 1985

[12] P. Bhugra, T.N. Mudge, Comparisons between Ada and LISP, U. Michigan, Research Report, 1985

[13] S. Reddy, F. Van Scoy, Knowledge Representation in Ada, Proceedings of the Eastern Simulation Conference, Orlando, 1987

[14] K. Warn, Lisp vs. Ada Implications in Diagnostics Oriented Expert Systems, Proceedings of AUTOTESTCON, 1986

[15] Terry B. Bollinger, Ada and PROLOG - A Few Observations, Proceedings of AIDA-87, Third Annual Conference on Artificial Intelligence and Ada, Washington D.C., 1987

[16] L. H. Reeker, K. Wauchope, Pattern-Directed Processing In Ada, IEEE Computer Society 2nd International Conference on Ada Applications and Environments. April 8-10, 1986, pp 49-56.